Fuzzy Logic and Musical Decisions

Peter Elsea, University of California, Santa Cruz,

Representation of Pitches as Sets

In the Max environment, pitches are necessarily represented as numbers, typically by the MIDI code required to produce that pitch on a synthesizer. We must begin with and return to this representation, but for the actual manipulation of pitch data other methods are desirable, methods that are reflective of the phenomena of octave and key.

A common first step is to translate the midi pitch number (mpn) into two numbers, representing pitch class (pc) and octave (oct) this is done with the formulas:

oct = mpn / 12
pc = mpn % 12

The eventual reconstruction of the mpn is done by

mpn = 12*oct + pc

In this system pc can take the values 0 - 11, in which 0 represents a C. Oct typically ranges from 0 to 10. Middle C, which is called C3 in the MIDI literature, and C4 by most musicians, is octave 5 after this conversion.

The major drawback of this representation is that it does not easily reflect the nearness of B (pc = 11) to C (pc = 0). Stating that C is above B is easy enough. The relationship:

pabove = (pc + 1)%12

holds true in all cases, but the reciprocal operation:

pbelow = pc-1

breaks when the zero boundary is crossed. The proper expression is counterintuitive:

pbelow = (pc+11)%12

The extra constraints of key and tonality are even more awkward to account for, and chords are difficult to manipulate.
A more flexible system of representation is the pitch set. A pitch set has twelve members, which are either 0 or 1:

1 0 0 0 0 0 0 0 0 0 0 0

The 1 represents the presence of the pitch in the set, which runs from C chromatically to B. This one is {C}. For any other pitch, the 1 is placed in the location that corresponds to the pitch.[1]. Any pitch can be produced by rotating {C} right by the desired interval[2].

The concept "pitch below" is produced by a left rotation, and is not upset by zero crossings, as:

1 0 0 0 0 0 0 0 0 0 0 0

produces:

0 0 0 0 0 0 0 0 0 0 0 1

The Max object that produces these sets is **Lror**[3], which will rotate left when given a negative rotation value. Lror is designed to default to the C pitch set, so any pitch set is easily produced by simply applying the pitch class to the left inlet of an empty Lror object.

Pitch sets can represent chords.

1 0 0 0 1 0 0 1 0 0 0 0

This set is a C major chord, and can be rotated to produce the other major chords.

The concepts of key and modality can also be represented by a set. This is the the C major scale:

1 0 1 0 1 1 0 1 0 1 0 1

All major scales can be produced from this by rotating (key) steps.

It is easy to exclude any pitch that does not belong to a key by a simple multiplication[4] between the scale set and the pitch set. Using this technique, we

[1] The first location in a Max list is location 0.

[2] Intervals are counted in half steps. A perfect fifth is an interval of 7.

[3] Many of the Max objects discussed, including all beginning with L, are from my expanded set of list processing objects. These are indicated in bold type the first time they are mentioned. They should be available from the same source as this paper.

[4] Any operation between sets is done member by member, so the pitch C survives multiplication between the C and Cmaj sets, but the pich C sharp would not. In set parlance, we are saying the the intersection of {Csharp} and {Cmaj} is empty.

can generate most chords in a given key. Start with a chordgen set, which is the union of the major and minor chords:

1 0 0 1 1 0 0 1 0 0 0 0

Rotate it to the desired position, say E (+4):

0 0 0 0 1 0 0 1 1 0 0 1

Finally, multiply it by the key set:

0 0 0 0 1 0 0 1 1 0 0 1
*
1 0 1 0 1 1 0 1 0 1 0 1
=
0 0 0 0 1 0 0 1 0 0 0 1

This produces an E minor chord.

These operations can be accomplished by two Max objects, Lror initialized with the chordgen set, and **Lmult**. (You will discover that this fails to generate the diminished chord on B. This problem can be solved with a rather complex patcher that, although interesting, is the beginning of a finally unproductive path. I will instead lay some more groundwork and return to the subject later.)

Any list of pitch classes can be converted to a pitch set by the **Ltoset** object. Sets may be converted back to pitch classes by the **Ltop** object, which reports the positions of the highest n values of a list. In the example above Ltop 3 would give the three chord members 4, 7, 11.

Crisp and Fuzzy Logic

The preceding exercise was really an example of Boolean logic. The multiplication of the scale set by the chord generator is equivalent to the logical AND operation, which gives the intersection of two sets. In traditional logic systems, an item is either a member of a set or it is not. G sharp is not a member of the C major scale, so it is represented by a 0 in the C major scale set. G is and gets a "truth value" of 1.

In Fuzzy Logic it is possible for items to have partial membership in a set. In other words, you might indicate a C minor scale like this:

1 0 1 1 0 1 0 1 1 0 0.7 0.6

Here the pitches C, D, E flat, F, G, and A flat are definitely members of the C minor scale. However, there are two different possibilities for the seventh degree. Some of the time B flat is used, sometimes B natural. A fractional membership value reflects these possibilities. Note that this is not a probability. That would imply that you knew how many lowered and how many raised sevenths there were going to be. These fractions merely indicate that either is possible, and that the rules for generating the pitches favor lowered sevenths somewhat[5].

Fuzzy logic makes it simple to represent concepts that are more linguistic than mathematical. For instance, in crisp logic[6], the concept "just below C" may be expressed by:

$$(x < 12) \;\&\&\; (x > 9)$$

This implies that we do not consider A to be just below C, and that B and B flat are equally just below C. But the usual meaning of the statement "just below C" implies a gradation of below C-ness that may include A in some circumstances. This can be represented with the fuzzy set:

0 0 0 0 0 0 0 0 0 0 0.2 0.5 1

Again, the membership values do not have to add up to anything, or fit any regular curve. They simply reflect our judgement of how well each pitch fits the criterion of "just below C". We can contrast this with the gentler constraint "below C" which might be represented by:

0 0 0 0 0 0 0 0 0.3 0.6 0.7 0.9 1

This is a bumpy curve that includes the feeling that there is a bigger jump between scale degrees than between the major and minor versions of the same degree.

The next linguistic step is to combine two descriptions. To find notes that belong to both sets "below C" and "in C minor" we find the intersection of the two sets. In fuzzy logic, intersections are most commonly found by taking the lower value of the equivalent members of each set. This is performed by the **Lmin** object.

The calculation of "below C in C minor" would yield:

1 0 1 1 0 1 0 1 1 0 0.7 0.6

[5] These fractions are basically made up to fit the situation. It is the relative strengths of the values that is important.

[6] Any logic that is not fuzzy is crisp.

Lmin

0 0 0 0 0 0 0 0.3 0.6 0.7 0.9 1

=

0 0 0 0 0 0 0 0.3 0.6 0 0.7 0.6

To get the result of the operation, you take the highest valued member with Ltop 1, in this case 10 (B flat). If you said "just below C in C minor"

1 0 1 1 0 1 0 1 1 0 0.7 0.6

Lmin

0 0 0 0 0 0 0 0 0 0 0.2 0.5 1

=

0 0 0 0 0 0 0 0 0 0 0 0.5 0.6

The result would be B natural.

Figure 1. is an example of this technique in use:

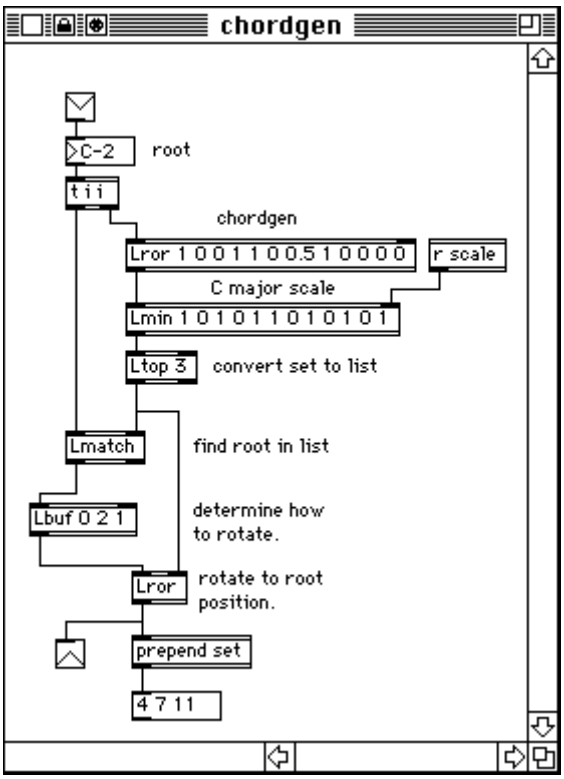

Figure 1.

The ChordGen patcher will produce the chords of the C major scale. To solve the problem of the diminished chord on seven, we modify the chordgen set to include the possibility of a diminished fifth,

1 0 0 1 1 0 0.5 1 0 0 0 0

rotate it to the B position and do the same calculation as before:

0 0 1 1 0 0.5 1 0 0 0 0 1

Lmin

1 0 1 0 1 1 0 1 0 1 0 1

=

0 0 1 0 0 0.5 0 0 0 0 0 1

This produces the pitches 2, 5, 11 as required.

The rest of the example is a chord sorter to leave the chord in root position. To do this I use **Lmatch** to find the position of the root in the pitch list. (Lmatch returns the position of one list or a constant within another list.) If no match is found there will be no output at the left outlet, so ChordGen will not produce chords foreign to C major.

The position of the root in the unmodified list is 0 for root position chords, 1 for second inversion, and 2 for first inversion.

To get these into the proper order I rotate them using instructions stored in the **Lbuf** object. Lbuf will return the value found at a particular position in a list.[7] In this case, the values are the ones necessary to rotate the chord into root order by the Lror object.

The ChordGen patcher will work with any scale.

Reasoning with Fuzzy Logic

The greatest advantage of fuzzy logic is the ease with which tasks may be translated into terms the computer can deal with. Most problems can be solved with mathematical models and advanced probability, but the construction of such models is difficult and the effective approaches are not often obvious. In addition, such models usually do not work at all until they are complete, and later addition of new factors can be a monumental task.

Fuzzy models, on the other hand, are a fairly straightforward translation of the linguistic statements of a group of rules. The model begins to function roughly as soon as two or three rules are stated, and is easily refined by tuning up the sets or by addition of more rules.

[7] Lbuf simply stores lists. It can be initialized with a list, which makes it easy to see what is happening.

In practical industrial applications, the fuzzy approach tends to lead to simpler, more easily maintained code in a shorter development time than other techniques.

An example of fuzzy reasoning.

To show how fuzzy procedures are applied to musical problems, I will continue with the issue of chord inversions. The choice of a chord inversion depends on many factors, the quality of sound desired, the avoidance of parallel fifths, fingering difficulties, and so forth. We begin the design process by formulating a set of rules as if...then... statements. Assume we want to choose inversions that will keep common tones where possible, that will not follow a root position with another root position, and will otherwise change inversions from time to time. The following rules state these criteria:

If root position keeps common tones, then root position.
If first inversion keeps common tones, then first inversion.
If second inversion keeps common tones, then second inversion.

If last position was root, then first inversion or second inversion.
If there have been too many firsts in a row, then root or second.
If there have been too many seconds in a row, then root or first.

The order in which the rules are listed makes no difference, as we are going to test all of them and combine the results. The final answer should be a number that can be used to rotate the chord to the desired inversion: 0 to produce a root, 2 to produce first inversion, 1 to produce second.

All of these rules have a predicate (if...) and a consequent (then...). We evaluate the predicate of each rule to find out whether to add the consequent into the combined result. If the predicate is crisp (as in "if last position was root") the consequent will either be reported or not.

In this example, each consequent is a set of three members. The value for member 0 is a vote for root position, the value in 1 is a vote for first inversion, and the value in 2 is a vote for second inversion. The consequent "then root or second inversion" will output the set {0 1 1}. The mechanism in figure 2 will do the work:

Figure 2.

If the predicate is fuzzy ("too many first inversions") the truth value extracted from the fuzzy set is used to modify the consequent in some way. One common modification is to clip the consequent set to the truth value obtained from the predicated rule; that is, make all members of the consequent set lower than the truth of the predicate. That is illustrated graphically by figure 3

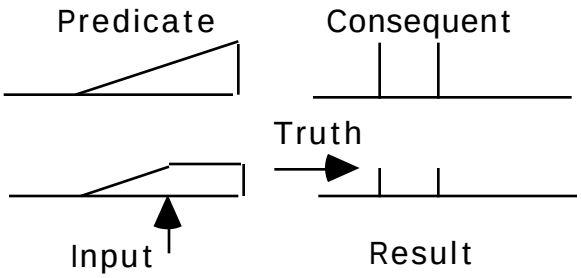

Figure 3.

The triangles[8] represent fuzzy sets for the predicate and the vertical lines the consequent. Some input value is used to derive a truth value from the predicate, which is then used to truncate the consequent. Figure 4 shows a Max mechanism to do this.

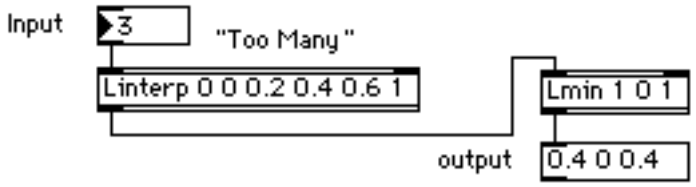

Figure 4.

In this operation, **Linterp** is used to find the value at a particular position in a set. Linterp can also find values between members by interpolation. That provides a lot more accuracy than these coarse examples suggest. Lmin, as we have seen gives intersections between sets. When the input to Lmin is a single value, the output set is truncated at that value.

Once all the rules have been evaluated, the answer is then derived from the accumulated consequences, the solution set. Many fuzzy applications involve fifty to a hundred rules, and the solution sets can get quite complex. Reducing these to an answer is called "defuzzification", and there are many approaches. One of the more common ones is to take the position of the maximum value produced in the solution set.

Figure 5 shows how these principles are applied to the problem of finding inversions.

[8] In Fuzzy Logic textbooks, consequent sets are represented by triangles too. In music the consequent sets are usually discontinuous, but the fuzzy techniques work just the same.

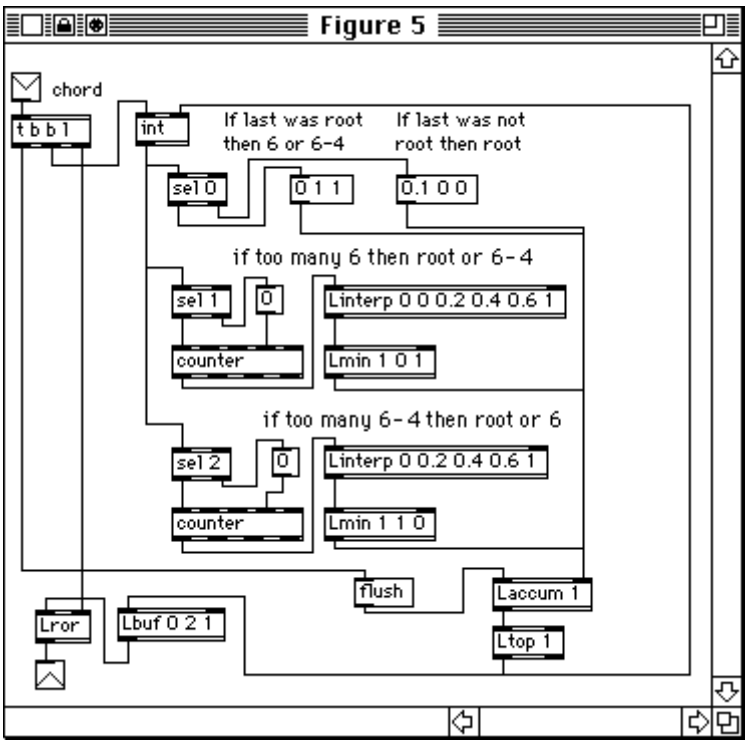

Figure 5.

The simplest rule to evaluate is the one about not repeating roots. In this case we can fire the result of the last cycle into the select object. If that result was 0 (root position), the list {0 1 1} which is the union of 6 and 6-4[9] will be sent to the accumulator.

The concept "too many" is definitely a fuzzy construct. It is represented by a set with members that describe how much "too many" each is. For instance

0 0 0.2 0.4 0.6 1

will allow two or three repeats easily. At 5 it will be pretty emphatically complaining.

We evaluate the predicate of the rule "too many first inversions" by looking in the "too many" set at the position that corresponds to the number of times the first inversion has occurred. That is monitored by a counter, as illustrated. Note that the counter is turned back to 0 by any other inversion. The Linterp object does the lookup.

[9] Note for those to whom music theory is a mystery: "6" is a common shorthand for first inversion chord, and "6-4" indicates second inversion. A root position C chord has the pitches in the order C-E-G (from the bottom up) or C-G-E. First inversion is E-G-C or E-C-G, and second is G-C-E, or G-E-C. The second version of each is called open position, and a complete inverter would cover those too. This one doesn't.

To produce the consequent of the rule, the result of the lookup is used to truncate the set "root or 6-4" {1 0 1}. The Lmin object will produce a set like {0.2 0 0.2}, taking the lower of each pair of values between the input and initialized set.

The rule for "too many 6-4" is identical except for the consequent.

The three result sets are added in the **Laccum** object, which is then flushed into Ltop 1 to reveal the winner. That value is transformed and fed back to another Lror (which is holding the chord in question) to produce the final output. The value is also saved in an int object for the next iteration.

Even at this early stage, the model will begin to function in a rudimentary way, flipping between root and first inversion. This allows testing of the control aspects of the patcher while it is still relatively uncomplicated. In this case, testing showed that a minor modification was needed to improve performance. An additional rule

> If last position was not root, then root.

insures that there is always an output from this part of the patcher. (It can be removed later when there is more information being processed.) It is given a very low weight, so that the main rules are not hindered.

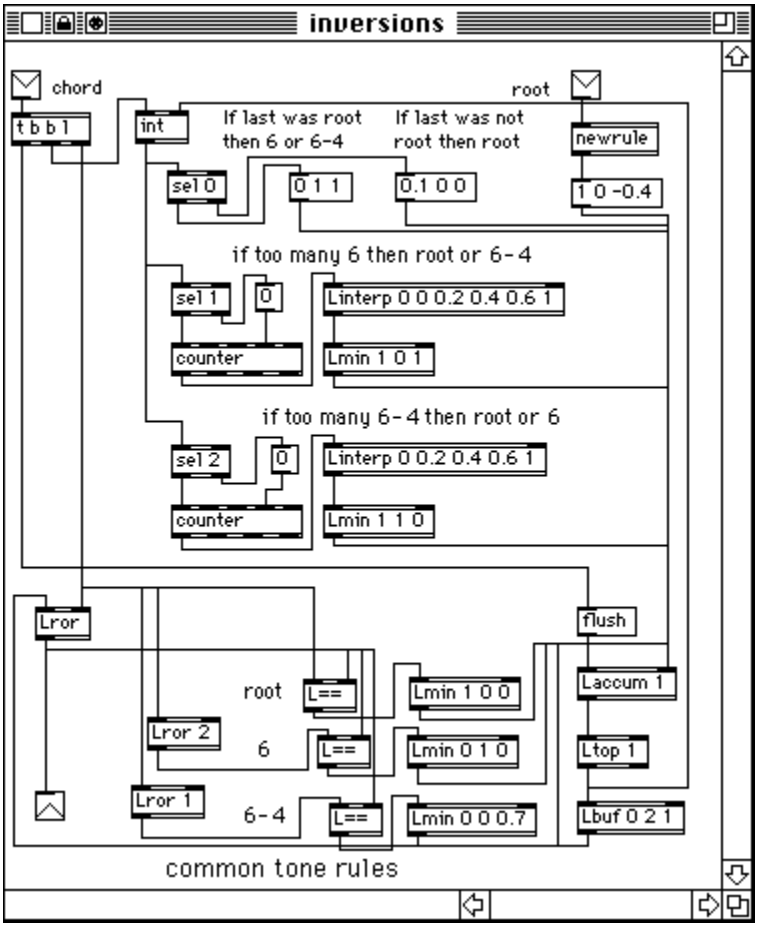

Figure 6

Figure 6 shows how the rules involving common tones were included in the patcher. The **L**== object compares two lists and returns a value between 0 and 1 that reflects the degree of similarity[10]. To decide if an inversion of a new chord will have any tones in common, we generate the inversion and compare it with the last chord output. If there are no common tones, the result will be 0. One common tone gives a 0.3, two 0.6, and if all tones are the same the output is 1. These weightings work well with the values produced by the previous inversion rules. Note that the consequent set for 6-4 is contains 0.7 instead of 1. This was edited to discourage second inversions slightly.

The newrule sub-patcher in the upper right corner of the inversions patcher shows the flexibility of the fuzzy logic methodology. Since evaluation is always based on a sum of rules, it is easy to add new ones. Experimentation with the working model showed that the progression V-I needed some special treatment. If the V chord happened to fall in root position, the tonic chord would be a 6-4, as the logic strives to keep common tones. The new rule simply

[10] It checks members of the input list against the same member of stored list. It then reports (number of matches)/(length of input).

detects a root of 7 followed by 0[11] and injects a set designed to skew the outcome toward root position.

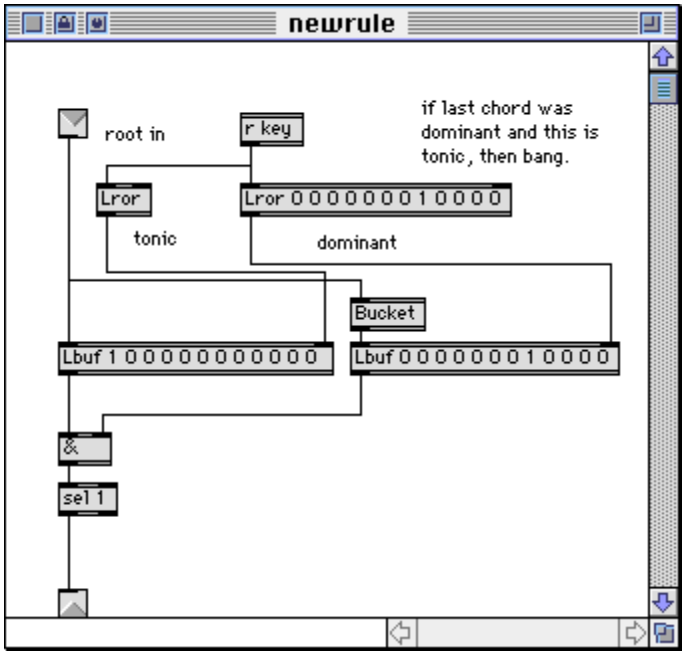

Figure 7.

This rule also illustrates how working with pitch sets can keep patchers simple. This crisp version of this patcher involves considerable mathematical gymnastics to allow for changing key.

At this point, the pitch classes are showing up in the output list in the desired order. The only thing left is to get them into the proper octave and play them. There are a variety of simple crisp ways of doing this, but I am going to indulge in a little more complexity to illustrate another point in fuzzy logic, as shown in figure 8.

[11] Remember, pitch class 7 is the fifth scale degree.

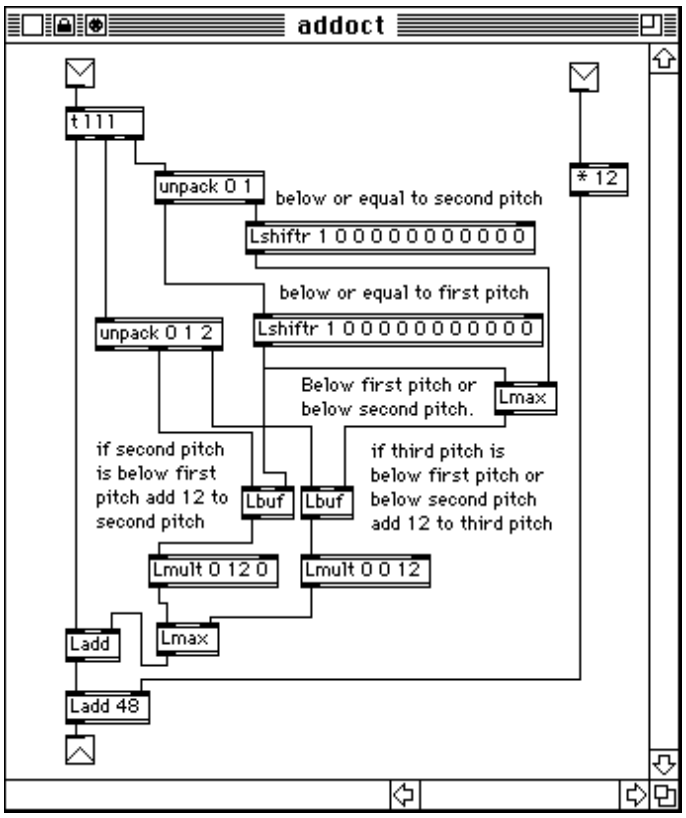

Figure 8.

The rules for deciding what value to add to the pitch class to wind up in the proper octave are simple:

> Add 48 (or another octave offset) to all three pitches.
> If the second pitch is equal or below the first pitch, add 12 to the second pitch.
> If the third pitch is equal or below the first pitch or the second pitch, add 12 to the third pitch.

Adding a constant to a all members of a list is simple with the Ladd object.

To generate a set for "equal or below a pitch" we use the **Lshiftr**[12] object on the C pitch set. Thus the set "equal or below E" is produced as in figure 9.

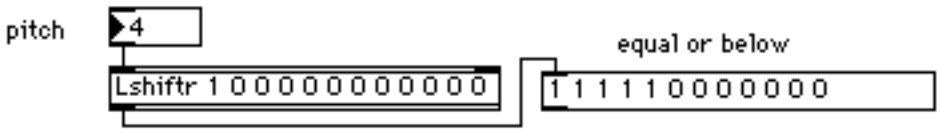

Figure 9.

[12] I realize the name is a little misleading, but the Lshiftr object shifts lists to the right. In the shift operation, all values are moved to the right, and values falling off the right end are discarded. The places freed up at the left end are filled with the leftmost value of the starting set.

This is fed into the Lbuf object, which will report a 1 if a requested pitch is equal to or below the first pitch.

We unpack the input list to apply the first and second pitch to Lshifr objects.[13] The output of the object labeled "below or equal to first pitch" is sufficient to test the second rule.

For the third rule, we need to evaluate "below the first or below the second". The word "or" implies a union. In fuzzy logic, unions are made by taking the greater value for any member in the two sets, here done by the **Lmax** object.[14]

We have already dealt with the and operation. The chordgen patcher described earlier really evaluates the rule "in the (root) chordgen set and in the Cmajor scale." As we have seen, "and" implies the intersection of sets and is accomplished with the Lmin object.

Using "or" or "and" to link simple tests, we can build rule predicates that are as complex as we wish.

To complete this operation, we evaluate each rule with Lbuf objects. Note that the list is unpacked again to produce the left inputs. This is necessary for timing purposes. If all of the pitch values were taken from the same unpack object, the rule for the third pitch would be evaluated before the union derived from the second and first pitches was constructed.

The results of these evaluations will be 0 or 1. They are multiplied by sets containing 12 in the appropriate position, and the union of the two resultant sets is added to the original list.

The output of the addoct patcher is a list of the pitches desired in the chord. This may be passed through an iter object and fed to makenote in the usual manner for producing notes in Max. However, more sophisticated performance is desired here. The criteria are,

New chords should curtail old chords.
Common tones between chords should not be re-articulated.
All tones should end after a preset duration.

[13] It is not really necessary to initialize the Lshftr objects with the C pitch set, they default that way. It is included here to make the operation clearer.

[14] These sets happen to be crisp, because below is taken in its literal sense, but if we were looking for the fuzzy concepts "just below C or just below G" with the sets described earlier, the **Lmax** method would also be appropriate.

As you have probably begun to gather by now, once you have stated the rules of a process, you are halfway to a working patcher. Figure 10. shows how the makechord patcher developed.

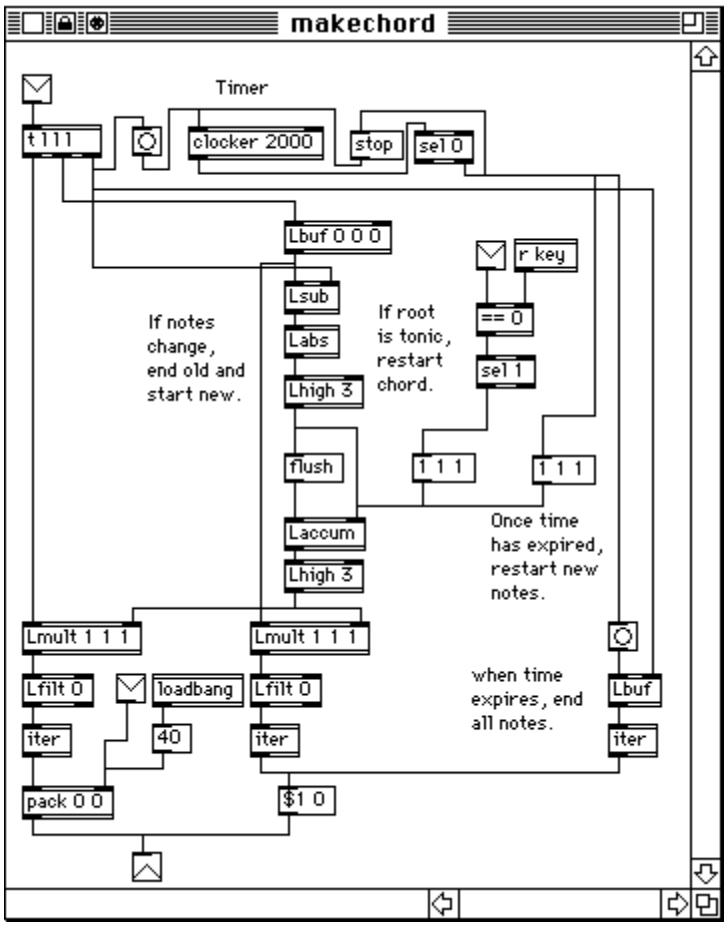

Figure 10.

The clocker and associated objects merely count the milliseconds since a note came in. If 2 seconds pass by, a bang is issued. This is a routine Max function.

The Lbuf object is used to detect any differences between the old chord and the new one. Up to now, I have been using Lbuf as a means for interrogating a particular member of a set. However, that is an incidental feature of Lbuf. Its main function is to store sets. If a list is input to the left inlet of Lbuf, the stored list is output and the new list is stored in its place[15].

To see which parts of the lists change, the last list is subtracted from the new list. Then the absolute values are found, and **Lhigh 3**[16] is used to convert all

[15] Thus Lbuf is really a Bucket for lists.

[16] The behavior of Lhigh could use some explaining. If applied to a list longer than its argument, it reduces all but the highest (arg) values to zero. If applied to a short list, it converts all non zero members to 1.

non-zero members of the result list to 1. This will create a control list that can be multiplied by the chords. The old chord is itered (after the 0s are removed) and each note is paired with a 0, which will create a note off when sent to note out. The new chord is treated the same way, but paired with a velocity value so a new note will be generated.

If the timer should happen to go off, the last chord played, which was stored in another Lbuf, is sent to the note off part of the output structure.

So far, the logic has been crisp. The fuzzy aspect of this patcher is simply that the outputs of various rules are accumulated to create the final control set. Again, that makes it easy to add features. In this case, the original criteria did not insure that if the chord had timed out the next chord would be played in full, so a new rule:

If time has expired, play all notes of the new chord.

was required. The timer simply feeds a control set of all ones into the Laccum. Likewise, the rule:

If root is tonic, play all notes.

was found desirable after some testing.

Figure 11 shows the master patcher. It takes notes in from a MIDI keyboard, applies all the processes discussed, and sends the chords out. There is obviously room for refinement, but even at this simple level it behaves in a very musical and unsurprising manner.

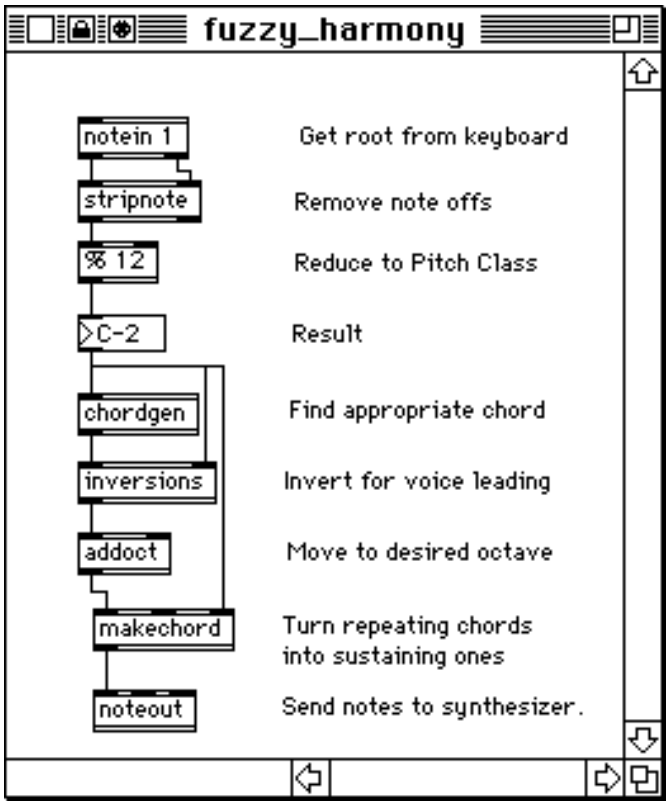

Figure 11.

Some more fuzzy concepts

Monotonic Reasoning

Many musical concepts are harder to express mathematically than would seem apparent at first blush. Take as an example, mezzo-forte (***mf***). Literally, this translates as medium-loud, a fuzzy concept if there ever was one. Musicians define ***mf*** as softer than ***f***, and louder than ***mp***, which is louder than ***p***.

If we consider that all these have a partial membership in the fuzzy set "loudness", we can place them in order and give them a membership that reflects common practice:

Loudness							
0.1	0.2	0.3	0.45	0.55	0.7	0.9	1
ppp	***pp***	***p***	***mp***	***mf***	***f***	***ff***	***fff***
			Dynamic				

Notice that the values do not grow evenly. In actual performance, the difference between ***mp*** and ***mf*** is not as great as that between ***mf*** and ***f***.[17] Also note that ***ppp*** does not have zero loudness, since zero loudness would imply silence.

The usefulness of the fuzzy approach becomes apparent when we want to perform a crescendo over n notes. We simply pick the starting and ending values and interpolate from one to the other through n steps.

The Linterp object will do this. It accepts fractional indices and reports a value interpolated from the previous and following entries. So if you feed 2, 2.2, 2.4, ...3 into

Linterp 0.1 0.2 0.3 0.45 0.55 0.7 0.9 1

You would get a smooth crescendo from ***p*** to ***mp***.[18] The advantage of this over a more direct approach, like assigning a velocity value to each dynamic and doing the math, is that this can be applied flexibly to various situations.

For instance, consider two instruments with differing velocity to loudness characteristics; one hits its loudest at velocity = 100 and the other maxes out at velocity = 120, with a non linear velocity curve. You can code these two curves into lists for the **Lfind** object:

[17] You are of course free to enter your own interpretation here, or to change values for different situations.

[18] If you crescendo from p to f you do not get a linear change, but it is an interesting and I dare say typical one. You may of course fiddle with this curve to suit your own needs.


```
Lfind 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
```

```
Lfind 0 0.05 0.1 0.15 0.2 0.3 0.4 0.6 0.7 0.85 0.9 0.95 1
```

Lfind searches a list for the input value and reports its position. If the value falls between two points, an interpolated fractional position is calculated. In this case, multiplying that position by 10 gives a velocity equivalent to the loudness desired[19]. We can use the loudness value calculated earlier to find the appropriate velocity to send to the desired instrument.

The curves in Lfind can be written in simplified notation. That is because Lfind expects a monotonic curve. If there are dips in the curve, a value may exist in two locations. Lfind gets around this by interpolating between the first value it finds that is lower than the desired value and the nearest higher value. Therefore zeros within a curve are ignored, and you only have to enter the inflection points[20].

This curve:

```
Lfind 0 0.05 0 0 0.2 0 0.4 0.6 0.7 0.85 0 0 1
```

would give the same results as the second of the pair above.

Linfer uses this simplified notation also. Given a list with two non-zero values, it will report an interpolated value for any intermediate position. An object like this:

```
Linfer 0.1 0 0 0 0 1 0 0 0.5 0 0 0 0 0.1
```

is a very simple envelope generator that can be reprogrammed by swapping lists around.

Fitting Vague Categories

One of the strongest features of fuzzy logic is that it allows classification of data within vague guidelines.

As an example, take the problem of finding dynamic markings for a group of notes input from a keyboard for transcription. The usual approach is to assign a range of velocities to each mark; 40-49 is piano, 50-59 mezzo piano, and so forth. The flaw in this method is that when group of notes cluster around the

[19] We have to shorten the list because the maximum size of a list in Lfind is 64. Besides, its much easier to read this way.

[20] This is know as singleton notation.

edge of a range the output will thrash between two markings. If the data for a phrase read 44, 51, 46, 58, 62, 67, 72, 74, most notes would get a marking, ***p*** , ***mp***., ***mf,*** or even ***f***. If we simply averaged the dynamics we would get 59.25, just over the edge into ***mf***.

The fuzzy approach is to assign every velocity a membership value in a set for each dynamic. In figure 12 the sides of the triangles represent the membership values[21] in each dynamic set for the velocities across the bottom.

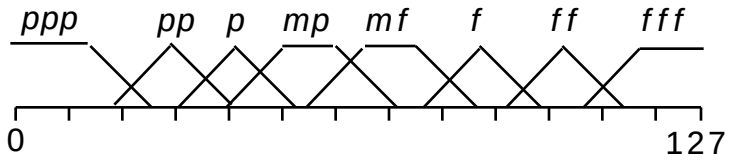

figure 12.

The sets overlap, so velocity 44 has both a membership value of 0.7 in *piano* and a membership value of 0.4 in *mezzo piano.*

The solution set for this problem will have eight members, one for each dynamic. The memberships in each dynamic set for each note in the phrase are added to the solution set, and then all members of the solution set are divided by 8 to get a set of averages:

0 0 0.15 0.52 0.48 0 0 0

The average amount of "pianoness" for our eight notes is 0.15. The average for ***mp*** is 0.52, and the average for ***mf*** is 0.48. The biggest average falls clearly within mezzo piano.

The fuzzy approach makes it easy to add more factors to the classification, such as giving more weight to notes near the beginning of a phrase. You would do this by finding a value for "near the beginning" from a set that looked something like Figure 13.

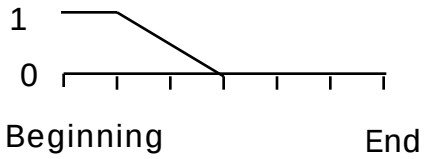

Figure 13.

[21] These are not the same numbers as in the previous example. Velocity from a given instrument will have a specific membership in each of the 8 sets for dynamic marks. Each dynamic has a membership in the set loudness. Loudness may be transformed to velocity for a particular instrument, but loudness should be determined by more factors than the dynamic marking.

This value would be used to scale the dynamic memberships for each note before adding them to the solution set. As you can see, the last few notes in a phrase would not affect the dynamic marking. For the next step, you could compare the starting dynamic with the dynamic found for the notes "near the end" and if necessary insert a crescendo or decrescendo.

Fuzzy Numbers

The fuzzy number is central to fuzzy logic and reasoning. Basically, a fuzzy number represents a concept similar to "approximately 5". It is a set of the type:

0 0 0 0.2 0.6 1 0.6 0.2 0 0 0 0 0

where the one is in the position corresponding to the number, and the shape of the flanking curve is appropriate to the application.

In music, intervals are fuzzy numbers. The concept "a second above" can have two possible answers depending on the scale and the starting point. The interval a second above is represented by the set:

0 1 1 0 0 0 0 0 0 0 0 0

We evaluate the complete construct "a second above D sharp in E major" by rotating "a second above" by three steps and then taking the intersection with the E major scale:

0 0 0 0 1 1 0 0 0 0 0 0

Lmin

0 1 0 1 1 0 1 0 1 1 0 1

result

0 0 0 0 1 0 0 0 0 0 0 0

This set may be evaluated by Ltop if we want the answer (E) or accumulated with other sets if we are building toward a more complex solution.

There is a complementary set for a second below

0 0 0 0 0 0 0 0 0 1 1

as well as above and below pairs for all of the imperfect intervals. These could be generated from the second by rotation, but it is probably most efficient to have them stored in a **coll** object.

The Edge of Fuzziness: Normalization

Fuzzy logic is a well defined discipline, and we should not play fast and loose with its conventions just because it deals with vagueness. However, some unique properties of music and the practicalities of the Max environment sometimes require us to do things that might raise an eyebrow in theoretical circles.

In fuzzy sets, all members are normalized to values from 0 to 1. This is an important convention, and must be observed if sets derived from a variety of processes are to be compared or merged. Generally, a normalization is performed after addition or subtraction of sets, and may be appropriate after intersection in some cases.

However, if the addition is the final step in an accumulation process that will then be evaluated by Ltop, normalization is not necessary. If a further intersection is to be performed on the results of the accumulation (which often saves processing steps) the Lmult object should be used instead of Lmin.

Another situation in which normalization is not needed is if the results will be passed to a table or other object that does not deal with floats. In that case the set needs to be multiplied by 10 or 100 and passed through **Lround** to insure that the values are in an appropriate range.

I have also found negative membership values to be of some use, particularly in the case of suppressing unwanted repeats. Throwing the last pitch into Laccum with a negative weight is a simple way of making repetitions unlikely.

Fuzzy Dice?

Fuzzy logic allows a vague description of the rules that will lead to a desired outcome, but it is still highly accurate and repeatable, given consistent initial conditions. However, in music composition, we do not always want the same outcomes, so we often add some indeterminacy to the program.

The simplest way to add this indeterminacy is to play roulette with the sets. Figure 14 shows a technique for doing this.

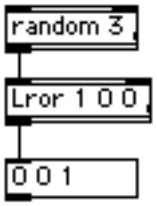

Figure 14.

The result of this operation, when accumulated with a group of rule evaluations, will randomly encourage one choice. The weighting of this random rule must be carefully chosen to maintain the character of the rule set.

A more powerful way of including indeterminacy is to load a set into a table. The **LtoTab** object is made specifically for this transformation, as it converts a list into a series of coordinated addresses and values that will transfer the list contents to the first locations in the table. Tables can only manage ints, so the set should be multiplied by ten or a hundred and rounded off.

The bang or quantile function of the table object will then output the address of one of the non zero members. The probability of getting any particular address is the value stored there over the sum of all nonzero values. So, if you loaded the set

5 0 0 0 2 0 0 3 0 0 0 0

into a table, 10 bangs should produce 5 Cs, 2 Es, and 3 Gs.

Indeterminacy with a higher level of organization can be achieved with a coll and the **unlist** object. The patcher in figure 15 processes the empty signal (a bang) from the unlist object to choose a list from the coll. Unlist then produces one member of the list at a time.

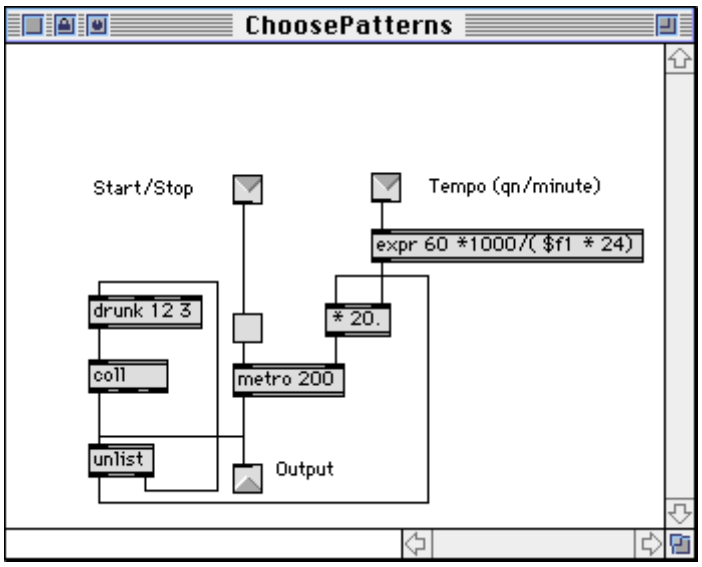

Figure 15

This patcher will randomly work its way through rhythm patterns stored in the coll object. The drunk object could be replaced by any rule evaluator, fuzzy or probabilistic.

The **Lchunk** and **Ltocoll** objects are useful for loading lists into coll objects.

Integrating Fuzzy and Probabilistic Reasoning

Indeterminacy in fuzzy composition systems is most effective when it is used judiciously at key points in the program. A simple composition engine may be modeled as in figure 16.

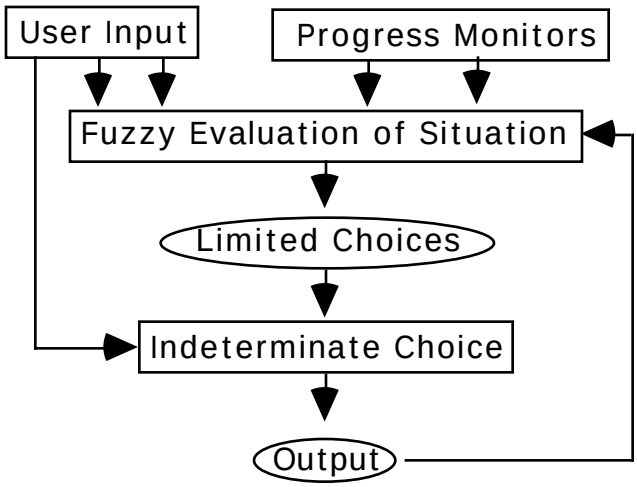

Figure 16.

This process is reiterative, with each output initiating and affecting the computation of the next. There are four major components that determine the results:

• User input includes initial settings as well as real time events. It is important that these inputs be accepted in terms the user understands.

• Progress monitors impose a temporal framework on the piece. This could be as simple as a metronome, or some complex phrasing system with its own user input.

• The Fuzzy evaluation rules incorporate the system's knowledge about the piece and music in general. They will provide desired amounts of predictability and self similarity, as well as conformance to the composers plan.

• The indeterminate section adds unpredictability.

Choosing from a set of reasonable options seems to me to be the essence of composition. Indeterminate choice is probably not the best way to do this, but until someone discovers an algorithm for creativity it will have to do. A practical system will have several of these composition engines, each working on some aspect of the music and all inter-linked.

The patcher in figure 17 is an example of such an engine, designed to produce fairly traditional chord progressions.

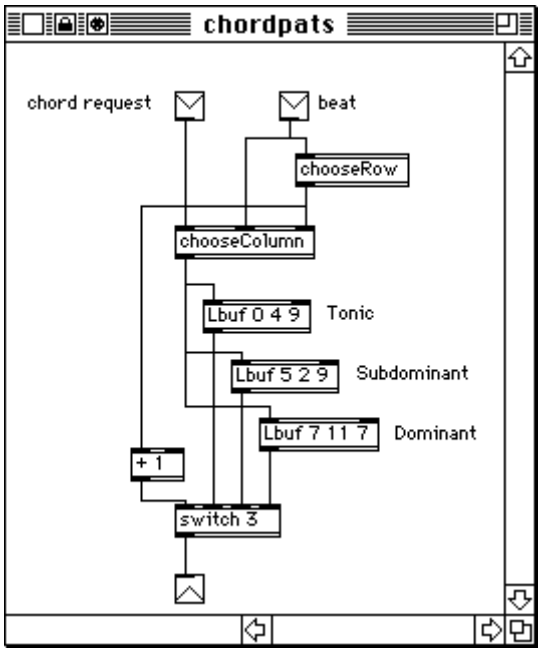

Figure 17.

The possible chords are grouped in the rows of a matrix according to function. The order of the chords in each row is suggestive of their order of probability. Translating the interval numbers into chord symbols, the matrix is really:

I	iii	vi
IV	ii	vi
V	vii	V

The chord will be determined by both the chooseRow and chooseColumn subpatchers. Working down the rows produces tonic, sub dominant, dominant, tonic progressions, moving across introduces variant chords in each function. The duplicate chords will not appear more often than the others because the rules within the subpatchers disfavor the third column.

The chooseRow patcher is shown in figure 18.

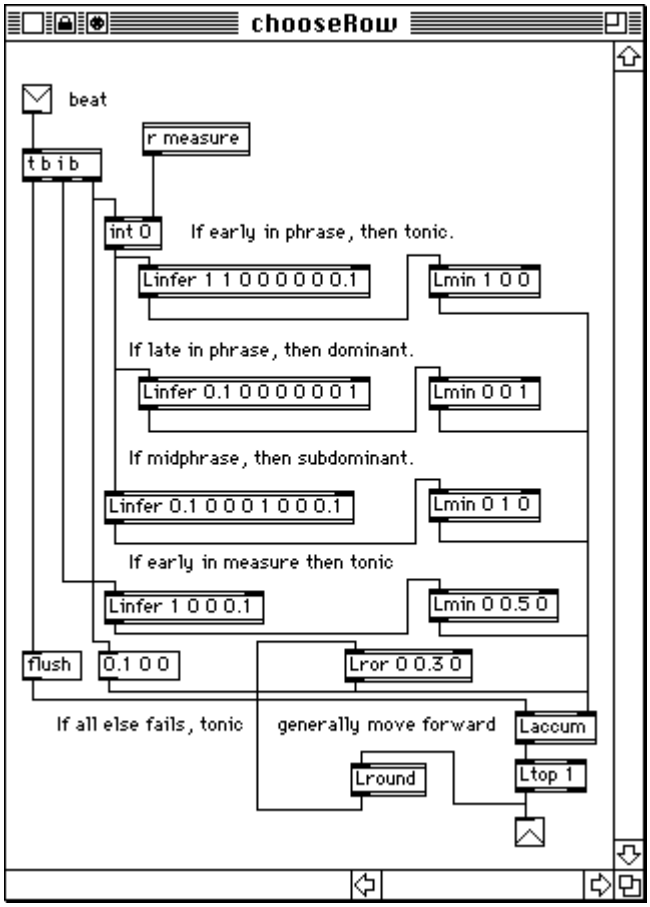

Figure 18.

This patcher uses fuzzy rules to choose a row according to the progress through the phrase. The choice is made on each beat, but the possibilities are controlled by the input from measure (here 0-7).

The three main rules:

If early in phrase then tonic
If mid-phrase then subdominant
If late in phrase then dominant

Will assert themselves strongly at times, but in the one-quarter and three-quarters progress points, the milder rules

Generally move forward
If all else fails then tonic

Will produce beat wise motion. These interactions are determined by the shape of the rule curves. Short, tall curves have high influence within a narrow region. Adjusting these curves (which could be done by user input) has a strong effect on the piece.

The chooseColumn patcher is shown in figure 19. It provides the indeterminant part of the puzzle, selecting one of two or three chords within each row.

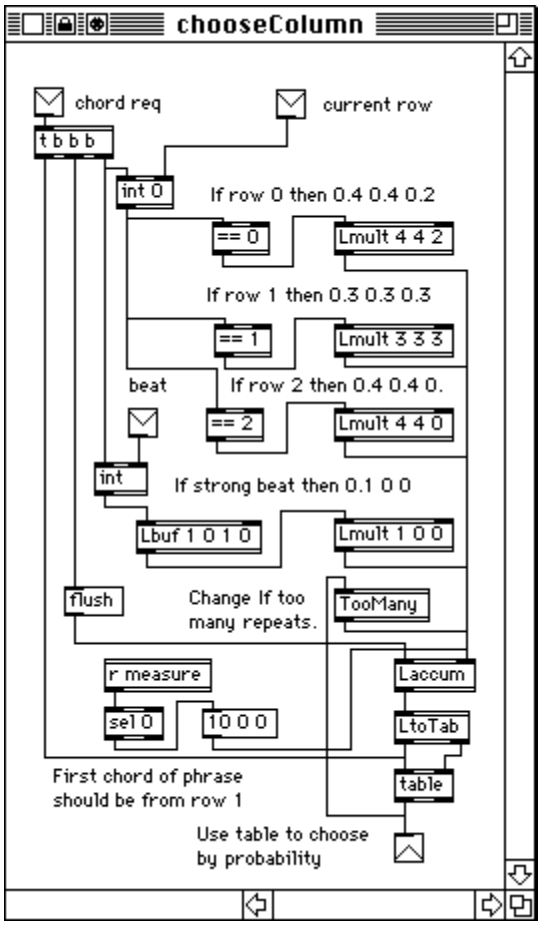

Figure 19.

The chooseColumn patcher uses the quantile function of the table object to make a probability weighted choice of columns. The basic probabilities are loaded into the table by a fuzzy mechanism, but the rules are crisp. The values chosen reflect my preferences at the moment. They are an excellent candidate for user input.

This is a classic Markov function that could easily be expanded to more dimensions of choice. A second order function can be achieved by using multiple predicates in the rules, such as:

> If row is 0 and last column was 0 then 0.2 0.4 0.4.

There is no reason the production of probabilities can't be fuzzy. You simply have to insure that all probabilities don't come out zero, or there will be no output from the table.

There are three additional rules in the chooseColumn patcher.

> On strong beats favor 1
> At beginning of phrase choose really favor 1
> If too many repeats, change columns

The TooMany rule is contained in a subpatcher, and works exactly like the too many repeats rule in the inversions patcher.

The entire structure is contained in the patcher shown in figure 20. This uses elements from the Fuzzy_Harmony patcher to play continuous arpeggios. The 12/8 time sub-patcher contains counters to produce beat and measure numbers within an 8 measure phrase.

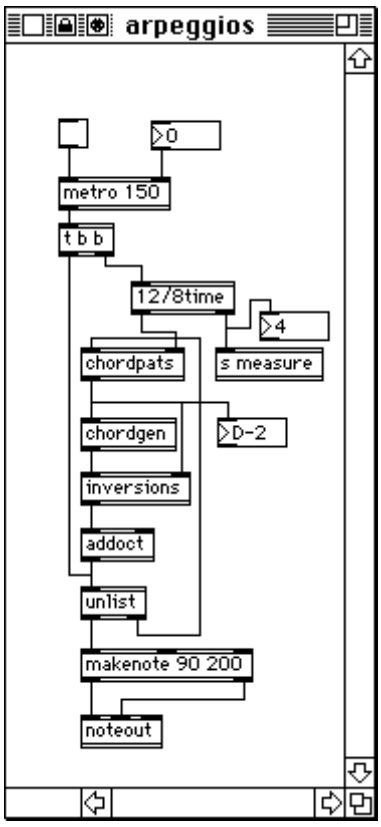

Figure 20.

Further Study

This tutorial has barely begun to explore the musical possibilities of Fuzzy Logic methodology. The ability of Fuzzy Logic to express traditional musical learning, and the ease with which Fuzzy procedures can be implemented, tested and expanded offer tremendous promise of a fruitful region of exploration. The examples offered here are literally my first experiments in Fuzzy technique, and represent only a few hours of development (mostly spent in tidying up the graphics). They are crude, but offer surprisingly sophisticated performance. I can foresee (as I hope you can by now) many problems that will benefit by the Fuzzy approach, and look forward to future developments.

This tutorial is also sadly deficient in expressing the principles of Fuzzy Logic itself. I leave that to better and more experienced writers. For beginning studies I can recommend:

McNeill, Daniel and Freiberger, Paul; Fuzzy Logic: The Revolutionary Computer Technology that is Changing our World. Simon and Schuster, 1993

And for practical application (including C++ examples and code on disk)

Cox, Earl; The Fuzzy Systems Handbook. AP Professional, 1994

